На предыдущем занятии мы с вами объявили базовый класс GeomBase геометрических фигур и дочерний от него класс Line прямых линий. Также узнали о новом режиме доступа protected. Давайте теперь поближе познакомимся с порядком работы объектов такого составного класса Line.

Пусть в функции main создается следующий объект:

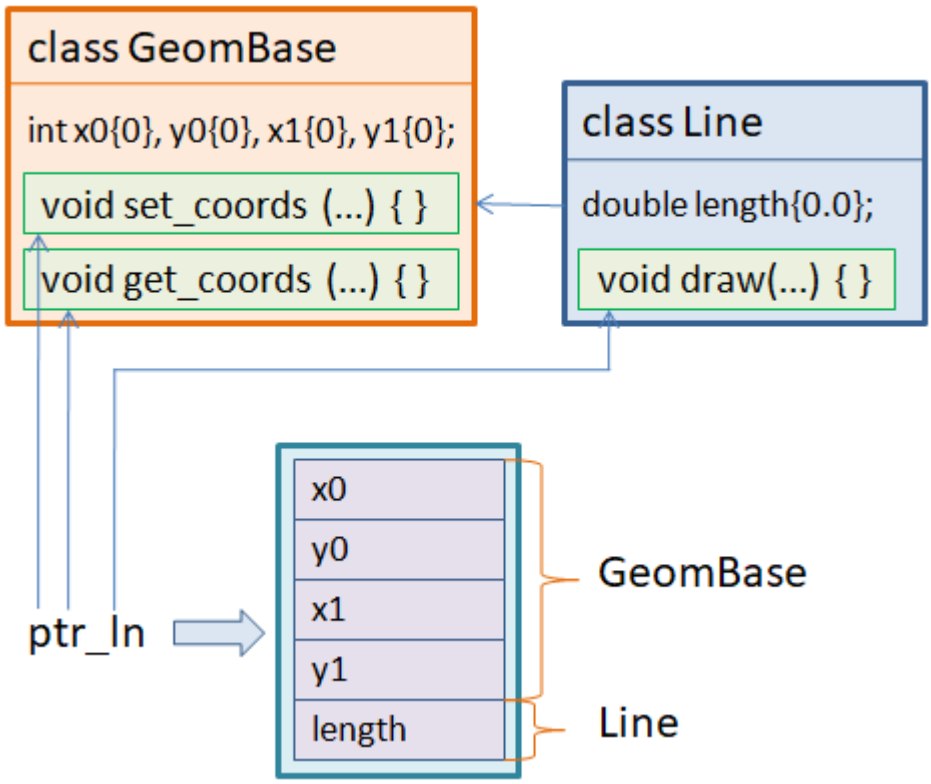
```language-cpp
int main()
{
    Line *ptr_ln = new Line;

    ptr_ln->set_coords(1, 2, 10, 20);
    ptr_ln->draw();

    delete ptr_ln;

    return 0;
}
```

Видим, что через него совершенно спокойно можно вызывать публичные методы, как базового класса (метод set_coords), так и дочернего (метод draw). Первый вопрос, что содержит объект, на который ссылается указатель ptr_ln?



В объекте, по-прежнему, будут находиться только данные, то есть переменные x0, y0, x1, y1, length. Причем порядок их расположения в памяти объекта будет именно такой: сначала переменные базового класса GeomBase, а сразу после – переменная length дочернего класса Line. Методов в объекте нет, они остаются на уровне классов и существуют в единственных экземплярах для всех объектов этих классов. Мы с вами об этом уже говорили. В частности, вызов:

```language-cpp
ptr_ln->set_coords(1, 2, 10, 20);
```

означает, что компилятор должен найти такой метод сначала в дочернем классе Line. **Если его там нет, то в базовом классе GeomBase и подставить его вызов с передачей в этот метод неявного указателя this. Причем тип этого указателя будет соответствовать типу объекта, из которого метод был вызван, то есть, тип Line, *а не GeomBase*, несмотря на то, что это метод базового класса**.

> Возможность вызова методов базового класса через объекты дочернего, называется **полиморфизмом адресов**._

Но для методов базового класса большого значения не имеет тип указателя. Главное, чтобы это был тип одного из его дочерних классов. В этом случае указатель, например, типа Line *всегда можно привести к типу базового класса GeomBase*, благодаря тому, что данные в объекте располагаются в порядке от базового класса к дочерним:

```language-cpp
    GeomBase* ptr_b = ptr_ln;

    int a, b, c, d;
    ptr_b->get_coords(a, b, c, d);
```

Причем, явно прописывать преобразование типов в этом случае не нужно. Это довольно частая ситуация в ООП и разработчик языка C++ решил сделать эту операцию неявной для удобства записи.

Сокрытие имен методов в дочерних классах

Давайте для примера в дочерний класс Line так же добавим метод set_coords с такой же сигнатурой, что и в базовом классе:

```language-cpp
class Line : public GeomBase {
private:
```

```
    double length{0.0};
public:
    void set_coords(int x0, int y0, int x1, int y1)
    {
        std::cout << "Line: set_coords()" << std::endl;
    }
...
};
```

Теперь, вызов:

```
ptr_ln->set_coords(1, 2, 10, 20);
```
language-cpp

будет связан именно с этим новым методом дочернего класса. Этот пример, во-первых, показывает, что мы совершенно спокойно можем переопределять методы базовых классов в дочерних. И, во-вторых, поиск методов ведется от дочерних классов к базовым.

Однако если мы сделаем вызов метода set_coords через указатель на базовый класс:

```
ptr_b->set_coords(1, 2, 10, 20);
```
language-cpp

то будет вызван метод именно базового класса.

А, что если, мы бы хотели сделать вызов какого-либо переопределенного метода именно из базового класса? Например, в методе set_coords дочернего класса можно было бы вызвать соответствующий метод базового, чтобы не повторять его логику. Сделать это можно, явно указав область видимости этого метода следующим образом:

```
    void set_coords(int x0, int y0, int x1, int y1)
    {
        GeomBase::set_coords(x0, y0, x1, y1);
        std::cout << "Line: set_coords()" << std::endl;
    }
```
language-cpp

И можно сделать даже так:

```
ptr_ln->GeomBase::set_coords(1, 2, 10, 20);
```
language-cpp

Мы через указатель на дочерний класс вызвали метод базового класса. Правда, в практике программирования использование последнего варианта,

скорее, будет говорить о неверной логике описания программы. Все же, встроенные правила и возможности ООП, как правило, ведут нас по верному пути проектирования программных архитектур.

Режимы наследования public, private и protected

Следует отметить, что вызывать публичные методы базового класса через объекты дочерних классов можно только в том случае, если используется режим наследования public:

```cpp
class Line : public GeomBase {...};
```

Если же ключевое слово public в наследовании классов не прописывать:

```cpp
class Line : GeomBase {...};
```

то это будет эквивалентно использованию ключевого слова private:

```cpp
class Line : private GeomBase {...};
```

Что в итоге изменится? Публичные методы базового класса по-прежнему можно будет вызывать в методах дочернего класса, но нельзя обращаться к ним извне через объект дочернего класса, например, так:

```cpp
ptr_ln->set_coords(1, 2, 10, 20); // ошибка из-за режима наследования private
```

А вот через указатель на базовый класс его публичные методы будут по-прежнему вызываться. Например:

```cpp
GeomBase* ptr_b = (GeomBase*)ptr_ln;
ptr_b->set_coords(1, 2, 10, 20);
```

Обратите внимание, что из-за режима наследования private компилятор теперь требует явного указания преобразования типа к базовому классу.

По аналогии работает режим наследования protected:

```cpp
class Line : protected GeomBase {...};
```

В этом случае все публичные атрибуты класса GeomBase при доступе к ним извне воспринимаются как protected со всеми вытекающими отсюда

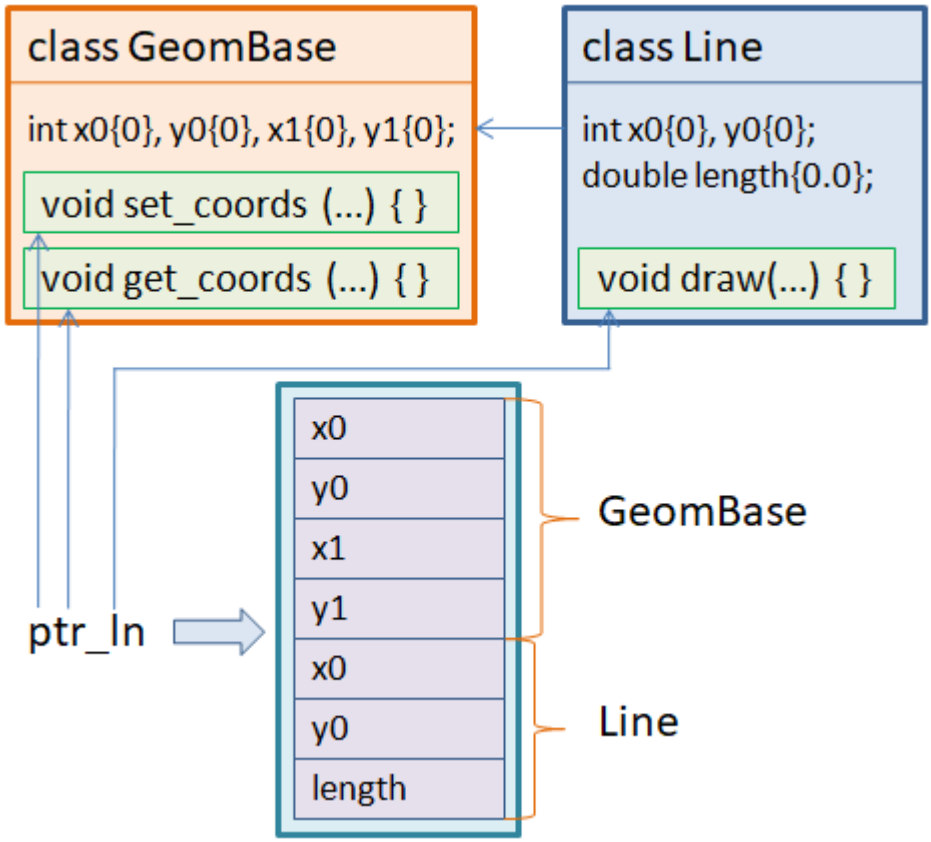
следствиями. Вот, по сути, отличия между режимами наследования public, private и protected.

Сокрытие переменных в дочерних классах

Наряду с методами также можно делать переопределение переменных в дочерних классах. Например, если в классе Line прописать переменные x0, y0:

```cpp
class Line : public GeomBase {
private:
    int x0{0}, y0{0};
    double length{0.0};
public:
...
};
```

то именно они по умолчанию будут использованы. При этом аналогичные переменные x0, y0 базового класса по-прежнему будут располагаться в самом начале объекта:



Мы легко можем в этом убедиться. Если выполнить программу:

```language-cpp
int main()
{
    Line *ptr_ln = new Line;

    ptr_ln->set_coords(1, 2, 10, 20);
    ptr_ln->draw();

    delete ptr_ln;

    return 0;
}
```

то в консоли отобразится:

```
Line: set_coords()
Line: 0, 0, 10, 20
```

Первые два нуля, как раз и говорят о том, что данные были записаны в переменные x0, y0 базового класса, а выводятся (с помощью метода draw) из дочернего, и в нем уже берутся свои значения x0, y0. Кроме того, если воспользоваться методом get_coords базового класса и прочитать значения его переменных:

```language-cpp
    int a, b, c, d;
    ptr_ln->get_coords(a, b, c, d);
    printf("Coords: %d, %d, %d, %d\n", a, b, c, d);
```

то в консоли появится строчка:

Coords: 1, 2, 10, 20

Видите, первые два значения уже не нулевые. Они были взяты из одноименных переменных x0, y0 базового класса, которые хранятся в текущем объекте.

Надо сказать, что на практике редко делают переопределение переменных в дочерних классах. Хотя, в ряде случаев это нужно, например, чтобы поменять значение какой-либо предопределенной константы. Но, как вы понимаете, злоупотреблять этим не стоит, т.к. переопределение переменных вносит путаницу в программный код.